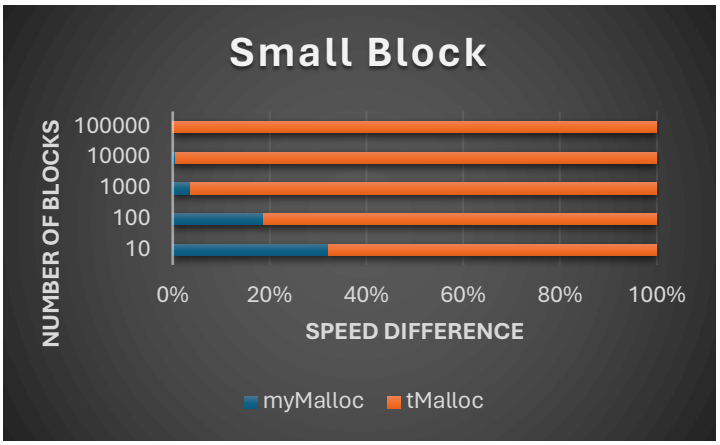
Data Report: Malloc Program
Benjamin Niccum
CSE 3320 Bakker UTA
03/14/2024

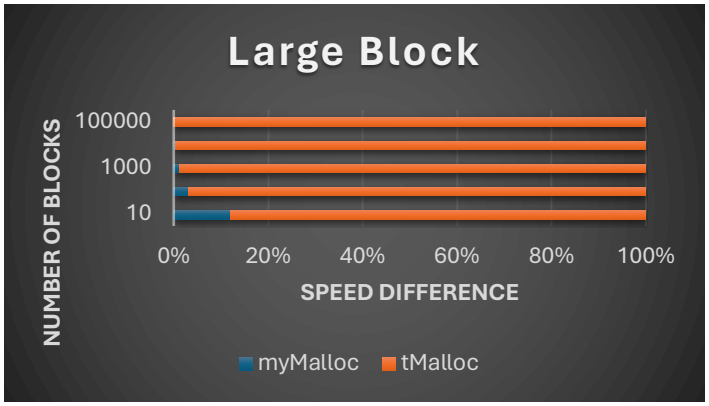
TABLE OF CONTENTS:

The test case that I wrote is like test one with several key differences. My Test (myTest.c) can be used with the library or used with the system standard malloc. The program is basic in that all it does is malloc blocks of memory and then free them. I added two variables to make the values for the size and number of the memory blocks dynamic. This allowed me to run different scenarios and measure the speed of each of the two Malloc versions. Then I added a time element to measure the speed of each run in ticks. A tick is approximately 100 nanoseconds. This is an adequate preciseness to measure the speed of most programs and gave me data necessary to compare Mallocs and different methods of memory allocation.

Using the test program that I designed, I ran the program 162 times to determine the speed difference between tMalloc and myMalloc. I used worst fit for each run in this set of trials. I could have used any of the four algorithms, but I needed to use one single algorithm throughout to not skew the results, and I happened to choose worst fit. First, I ran the program with different numbers of blocks starting with ten blocks and ending with 100,000. For this set of tests, I used a block size of 100. I ran each number 20 times and took the average of those results apart from N=100,000 on myMalloc which I only ran once because it took a very long time to return the result. I then repeated the experiment with the block size set to 5000 instead of 100 so that I could compare speeds of

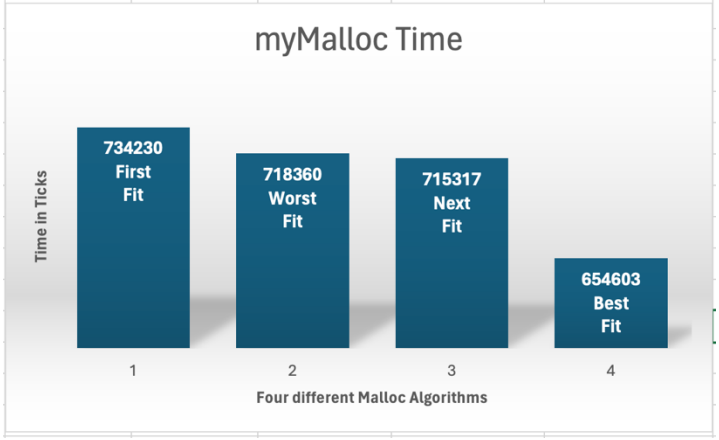
All run with Worst Fit				
BlockSize	N	tMalloc Time	myMalloc Time	speed difference
100	10	8.05	16.95	-53%
100	100	25.3	109.05	-77%
100	1000	189.82	5005.18	-96%
100	10000	2013.91	405031	-100%
100	100000	17552.45	42585293	-100%
	Average	19790	8599091	
5000	10	6.45	46.91	-86%
5000	100	16.92	516.46	-97%
5000	1000	114.27	8972.64	-99%
5000	10000	2206.64	732496	-100%
5000	100000	14136.27	71726550	-100%
	Average	16481	14493716	





different block size. The data table shows the results of my tests. Using this data, I created the two graphs that show the speed difference between myMalloc and tMalloc using small blocks and large blocks. The results were surprising. Traditional malloc performed better with larger blocks, and my Malloc performed worse with larger blocks. You can tell this by the sum of averages at the bottom of each table in the data table. The main determinant being the speed with 100,000 allocations for each set. My presumption would be that the results would be similar even with different algorithms since they do basically the same thing, but this was not the case. From the two graphs you can determine that tMalloc performs considerably better than myMalloc regardless of block size or number of allocations. I reviewed the code library for traditional malloc, and I can tell you that my Malloc is nowhere near as complicated or documented as the traditional version, and for their hard work they get to say they have the better version. My tests did show that to be the case. At higher numbers of allocations (N), tMalloc performs 99.99% better than my worst fit algorithm, but at lower values of N it only outperforms my algorithm by about 80-90%.

Four Malloc Algorithms				
	Block Size	N	myMalloc Time	speed difference
First Fit	2500	10000	734230	112%
Worst Fit	2500	10000	718360	110%
Next Fit	2500	10000	715317	109%
Best Fit	2500	10000	654603	0%



For the next set of tests, I was not able to use traditional Malloc because I wanted to test the four algorithm options in my Malloc. I ran each algorithm 20 times with the blocksize at 2500 and the number of allocations at 10,000 and took the average run time measured in ticks using the non-traditional Malloc library. The data and graph table show the results of these test runs. Best fit was indeed the best fit. This algorithm starts at the first block of memory in a series of memory blocks that may be in use or may be free. Best fit searches

through all the blocks in a set size of memory. If it finds a block of memory that will hold the requested blocksize, it saves that information and continues. Then if it finds a better block it saves that. Better in this case is defined as large enough to hold the block, but as close to the size of the block as possible. Once it reaches the end, if it has not found an acceptable block, it will grow the search area and try again. If it does find a good block, or several good blocks, it will choose the best block and return it. All four of the algorithms share some aspects of this process and so I will only explain the differences instead of explaining every algorithm in detail. For the data table I set Best Fit at 0% and compared the other three algorithms to Best Fit for comparison. The simplest algorithm was the slowest performer, and that algorithm is First Fit. First Fit does not optimize the fit in the way that Best Fit does. It simply searches the blocks until it finds a spot in memory that is big enough and free, and then it returns that block. Best Fit was 12% faster than First Fit. Next Fit is like First Fit, but it remembers where First fit left off, and then continues from that point until it finds the next acceptable block and returns that block of memory. This algorithm performed 10% slower than Best Fit. Finally, performing at 9% slower than Best Fit, we have Worst Fit. Worst Fit works almost exactly like Best fit, but instead of remembering the Smallest Block that would be acceptable, it remembers the largest block. Intuitively, this method would result in a great loss of memory if I had not also written functions included in each of the four algorithms that split one block into two and leave the remaining block, or in some cases coalesce two blocks into one. These methods were required to optimize available memory space.